Gabrielle Maitland

CS 320 Software Test Automation and QA

**Project Two**

In my testing approach, I aligned to the software requirements by testing individually for every specification. An example of this can be seen as early as Module Three where we had to write a program that featured variables that could be deleted, updated, and of a certain length. I made sure to write tests that could prove that I achieved the requirements. In ContactTest and other files thereon, I used the Assertion function that Java provides in order to verify that those fields were indeed able to be updated, deleted, and of the correct character limit.

In previous Modules, my Junit tests did not yield high coverage percentage. This was due to not writing enough test cases for each branch of logic. For example, for lines that included a true or false Boolean, I only tested to ensure that an exception error was thrown for the false branch, but that left the true branch untested. I remedied that by checking the detailed coverage report to see which lines were only being half tested or not tested at all. This allowed me to raise my overall coverage percentage to 88%.

I ensured that my code was technically sound by double checking my code even after I wrote it. Even though the “Problems” tab in Eclipse didn’t show any warnings for errors or potential issues, it not a definitive conclusion that there are not any uncaught syntax errors or otherwise. I was previously having trouble with my ContactServiceTest file in which every test I ran was failing. It was difficult for me to identify why at first as there was nothing glaringly obvious. Even after applying feedback from that week’s work, I still couldn’t get any test to pass. I revisited the file a few days later to investigate the problem more in depth and I was able to find the source of the issue. I used the wrong syntax due to omitting an additional | for logical or. Once I corrected the afflicted lines, my tests passed.

I ensured efficiency in my code by trying not to be too verbose. Sometimes as a new developer, it is easy to take a very long and round about way to achieve a goal, generating multiple lines of code and potentially a longer run time. However, I tried to keep it simple by combining some of my branching logic rather than writing them out separately, as seen in Contact.java lines 12-25.

In this project, I used both static and dynamic testing techniques for each of the specifications. To test the executable statements and logic branching in my code, I implemented statement testing and decision testing. Static testing was used to clean up and clarify declared variables that may have otherwise been ambiguous. Functions were contained to a few lines to avoid writing excess code. Dynamic testing was used when running my code to reveal any errors in compilation and testing. As aforementioned, I ran into some instances in which the “Problems” viewer did not immediately alert me to any issues. However, there were instances in which I wrote test cases to assert true but used the wrong parameter and ended up with false results.

There are many more forms of testing techniques that we learned about, however I did not have uses for all of them. Some of the techniques that I did not use include parametrized tests, non functional tests, and mocking. Parametrized testing is a great technique, as it allows testers to exercise different parts of their code against multiple values. Non-functional testing is used to test for system traits like efficiency, performance, and usability, amongst others. Mocking can be used to increase coverage rates for Junit tests. In a more practical sense, if I wanted to test for behavioral categories of a system, I’d implement non-functional tests. If I had a bunch of different values I wanted to test against a section of my code, I could write parametrized tests to achieve that rather than writing multiple lines. If testing coverage was a huge requirement for a project before it went live, I would implement mocking to achieve higher rates.

In retrospect, I did not realize how much caution I employed when writing the code. However, I’m not sure if I would call it a cautious mindset or a detail oriented mindset. I think the two go hand in hand in a lot of cases. I paid attention to make sure that I was calling the right variables and that I was wording my test cases properly to avoid mishaps, especially since there were interdependent files written. Each time I received feedback, especially early on, I would go back to the previous weeks and change things to meet the criteria set.

I tried to limit bias by writing extra test cases and not just submitting the bare minimum. This helped to raise my coverage percentage greatly. It is easy to run your tests, see that they pass, and move on – especially when you have so much other work to do. Yet I had to have a pep talk with myself to set aside a little more time to actually assess what the test cases looked like and what I could add. Bias is in all of us in some way, shape, or form. The difference is how self aware you are and what you do to combat your own biases. This is true in all areas of life, especially personally.

It is very important to be disciplined in your commitment to quality as you can easily get comfortable turning in subpar work. It is one thing to still be in the novice stage and figuring out how to improve your work, but it is another thing to have a few years of experience and to still choose the bare minimum at all times. Cutting corners when writing and testing code causes problems down the line for everyone else, which makes for an unnecessarily stressful work environment. In my very little experience, I practice avoiding technical debt by consistently checking in with my seniors. I make it a point to try as much as I can, read through the docs and other resources, and then come to them with my question and troubleshooting steps. This happens on a regular cycle for me weekly. This helps to finetune your code with an experienced outside perspective who can immediately identify what you’re lacking. Because I did that so often, it became easier for me to see what areas I was missing that could potentially cause more work for others later.

**Resources:**

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing : An istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.

Maitland, G. (2023). *Module Two Journal* [Journal Assignment]. Southern New Hampshire University.

Maitland, G. (2023). *Module Four Journal* [Journal Assignment]. Southern New Hampshire University.

Maitland, G. (2023). *Module Five Journal* [Journal Assignment]. Southern New Hampshire University.